Coarse-Grain Parallel Computing Using the ISIS Toolkit

By Ralph Finch[1], Associate Member, ASCE, and Shao-Kong Kao[2]

Abstract: A coarse-grained parallel (distributed) computing application using the ISIS toolkit is described. The method should be applicable to any serial program suitable for coarse-grain parallelization. Criteria for parallelizing existing programs, factors which affect the speed of parallel programs, and benefits in addition to quicker turnaround time are described. Future tasks to obtain more benefits are discussed.

Introduction

Greater computer speeds have become available to the computer modeler as computer prices have fallen on faster computers. This increased computer availability, instead of satisfying demand, has in many cases only fueled the clamor for greater throughput and shorter turn-around times.

Traditionally, quicker model runs have been possible through only one avenue: buying faster hardware which uses a faster clock rate. In recent years, however, different forms of parallel computing have been used to decrease the elapsed time from start to finish of a model run, by splitting apart the program and running the different pieces simultaneously. These forms of parallel processing are generally classified according to their granularity, that is, how big a chunk of work is done in a parallel fashion. At one extreme lies vectorization, where a series of a few atomic arithmetic units (multiplies, adds, etc.) are done in parallel, and each parallel task might take only a few clock cycles or microseconds to complete. At the other extreme lies distributed computing, where entire subprograms are done in parallel (usually across a network of computers), and each task takes several minutes or hours to complete.

This paper reports a distributed computing application which used to run in the normal fashion in serial on a single machine, and took several days to complete. Described is how to identify an application as suitable for parallelization; why the ISIS toolkit was picked as the parallelizing software; how the application was converted from a serial one to a parallel one; and the benefits reaped from the parallel application. Although the authors' particular

[1] Senior Engineer, W.R., Calif. Dept. of Water Resources, Div. of Planning, 1416 9th Street, Sacramento, CA 95814

[2] Formerly, Ph.D. student, University of California, Davis. Now with Capital Marketing Technology, Inc., 1995 University Ave., Berkeley, CA 94704.

application will be described here, the authors wish to emphasize that any suitable application can be parallelized, not just this particular problem.

The Application

The first author works in the Modeling Support Branch in the California Department of Water Resources. This branch of the Division of Planning is responsible for modeling hydrodynamics and quality in the Sacramento-San Joaquin Delta (WEF, 1989; Bulletin 160-87, 1987). Primarily two models are used to accomplish this task: the DWR Delta Simulation Model/Hydrodynamics, and DWRDSM/Quality, both derived from the Fischer Delta Model (Fischer Inc., 1984). In the classification scheme of mathematical models given by Karplus (Karplus, 1983), these are both continuous space—continuous time models, and fall somewhat in the gray region: they are not entirely transparent or white, as electrical circuit models often are, nor are they opaque or black as sociological or political models are. As such, while the equations and relationships describing flow and mass transport are well known, their parameters are not known exactly and must be fitted or calibrated to a set of observed excitations (inputs) and responses (outputs). In the case of the hydrodynamics model, the inputs are channel stages and flows at the boundaries and internal sources, and the outputs are stages and flows in all channels. In the case of the quality model, inputs are salinity concentrations at the boundaries and internal sources and the outputs are concentrations in all channels. Calibration of the models are done by specifying both the inputs and outputs and adjusting channel parameters so as to reduce the error between measured and computed outputs. For the hydrodynamics model the Manning's n channel roughness coefficients are adjusted, and for the quality model the dispersion coefficients.

In the past, calibration was done by hand (Fischer Inc., 1984). More recently, the Department has been using the Parameter Estimation Program (PEP), developed by Water Resources Engineers, Inc. for the Office of Water Research and Technology, US Department of the Interior (Evenson and Johnson, 1975). To quote from the users' manual, "... [PEP] automatically estimates model parameters to minimize the error between simulated variables and field measurements." That is, minimize

$$\boldsymbol{P}(\boldsymbol{K}) = \sum_{t=1}^{\tau} \epsilon_t \epsilon_t^T \qquad \textbf{(1)}$$

where ***P(K)*** is the performance function; $\epsilon_t = \hat{h}_t - \bar{h}_t(K)$, a column vector of ***N*** calibration errors at each measurement location $(\epsilon_{t1}, \epsilon_{t2}, \ldots \epsilon_{tn}, \ldots \epsilon_{tN})$ at time ***t***; $\hat{h}_t$ is a column vector of ***N*** measured variables at time ***t***; and $\bar{h}_t(K)$ is a column vector of ***N*** computed variables at time ***t*** which are a function of the parameters to be calibrated ***K***; and superscript *T* indicates the transpose of the vector. To obtain the minimum of Eq. 1 the partial derivative is taken and set to zero:
When Eq. 1 is differentiated,
where $\boldsymbol{J}_t$ is an ***N*** x ***L*** dimensional Jacobian matrix defined at time ***t:***

$$\frac{\partial P(K)}{\partial K_l}=0 \qquad l=1, \ldots L \tag{2}$$

$$\frac{\partial P}{\partial K}=-2\sum_{t=1}^{\tau} J_t^T[\hat{h}_t-\bar{h}_t(K)]=0 \tag{3}$$

$$J_t=\begin{bmatrix} \frac{\partial \bar{h}_{t1}}{\partial K_1} & \cdots & \frac{\partial \bar{h}_{t1}}{\partial K_N} \\ \vdots & & \vdots \\ \frac{\partial \bar{h}_{tL}}{\partial K_1} & \cdots & \frac{\partial \bar{h}_{tL}}{\partial K_N} \end{bmatrix} \tag{4}$$

Gauss's linearization technique is applied to Eq. 3 where a Taylor series expansion of the iteration ***i*** value of $\bar{h}_t$ is taken about iteration ***i-1*** values of ***K***. When second and higher order terms are neglected, this results in:

$$\bar{h}_t^{(i)}(K)=\bar{h}_t^{(i-1)}(K)+J_t^{(i-1)}\Delta K \tag{5}$$

If the correction $\Delta K=K^{(i)}-K^{(i-1)}$ is small, $J_t^{(i)}\approx J_t^{(i-1)}$ and Eq. 5 may be substituted into Eq. 3 to yield:

$$A^{(i-1)}\Delta K=B^{(i-1)} \tag{6}$$

where

$$A^{(i-1)}=\sum_{t=1}^{\tau} J_t^T J_t \tag{7}$$

$$B^{(i-1)}=\sum_{t-1}^{\tau} J_t^T \epsilon_t^{(i-1)} \tag{8}$$

The partial first derivatives are computed numerically by slightly perturbing the parameter for channel group ***l***, then running the model with all other channels unperturbed. Since there may be anywhere from 5 to 30 groups, the calculation time for each group is from 5 to 120 minutes on a Sun 4, and from 6 to 12 iterations must be performed to converge to a calibrated set of parameters, total running time could be as long as 30 days when run in a serial mode. In practice, the number of groups is reduced, a shorter time span used for the calibration period, and few iterations done, resulting in a less accurate calibration with still

lengthy running times of up to 10 days. Thus there is an obvious motivation to reduce elapsed time by significant amounts, as opposed to marginal improvements of 10-20%.

Criteria for Parallelizing a Program

The key to parallelizing any program is that the tasks which are to be run simultaneously must not depend on each other; that is, their inputs are not a function of the other tasks' output. In the application described here, it is apparent that the calculation of any given numerical first derivative $\frac{\partial \bar{h}}{\partial K_l}$ is independent of the calculation of the others. Thus, the problem is amenable to some form of parallelization.

Furthermore, a significant fraction of the program must be parallelizable, and the granularity of the program must be suited to the overhead of the parallelizing method. This will be explored further below.

Considerations in Parallelization

Naturally the first question that one might ask before starting a project to parallelize an existing program is, how much will the program elapsed time decrease? And, what factors would affect the speedup? Here, the speedup is defined as the ratio of the elapsed time to run the serial program to the elapsed time of the parallel version of the program:

$$S = \frac{E_s}{E_p} \tag{9}$$

where S is the speedup of the parallel program over the serial program, E_s is the elapsed time of the serial program, and E_p is the elapsed time of the parallelized program. In general, not all of a program will be able to run in parallel, thus E_p may be calculated as:

$$E_p = E_s - fE_s + E_p' \tag{10}$$

where f is the fraction of the program that can be parallelized (measured by processor time, not by amount of code), and E_p' is the elapsed time of a parallel job (it is assumed that all parallel jobs are of equal length). This is simply the parallel version of Amdahl's Law (Amdahl, 1967).

The parallel portion of the program is composed of N_j equal length jobs, to be run on N_p processors, $N_p \leq N_j$, and N_p and N_j integers. Also considered is the communication overhead to pass messages and data between parallel jobs and the controlling routine, so that

$$E_p' = (\pi_j + \pi_c)\frac{N_j}{N_p} \tag{11}$$

where $\pi_j = \frac{fE_s}{N_j}$ is the processor time of a parallel job (computational overhead) and π_c is the processor time needed for communicating between parallel tasks (communication overhead).

Substituting Eq. 10 and 11 into Eq. 9 yields

$$S = \frac{1}{1 + f(\frac{1}{N_p} - 1) + \frac{\pi_c N_j}{E_s N_p}} \tag{12}$$

Eq. 12 is a variation on the first-order parametric model of parallel computing given by Hack (Hack, 1989). Basically there are three possibilities open to increasing the speedup: increasing the fraction of the program that can be parallelized, decreasing the communication overhead, and increasing the granularity of the parallel portion of the program in conjunction with more processors. Each one of these possibilities is discussed further below.

Increasing the fraction of the program that can be parallelized, perhaps by using different algorithms or recoding, is quite useful if it can be done. Figure 1 shows a plot of fraction f versus speedup with typical values for the other variables for some hypothetical program. To paraphrase Buzbee (Buzbee, 1986), a little parallelization does not go a long way toward improving performance; a significant fraction of the program must be run in parallel to obtain worthwhile speedups. It can also be seen that with more parallel jobs (decreased elapsed time of the parallel portion), it is even more important to be able to parallelize a greater fraction of the program.

Communications overhead becomes a factor when it approaches the magnitude of the computational expense, and can be reduced by using different parallelizing software or hardware. Figure 2 shows a plot of the ratio π_c/π_j vs speedup, again using a hypothetical case. In our application the ratio is extremely small and thus can be ignored, but for fine-grained applications this can be an important factor.

Finally, one must consider the possibility of increasing the granularity of the parallelizable portion of the program and adding more processors to the system, thereby decreasing the time E_p'. Figure 3 shows the benefits of this approach with the granularity defined as the number of jobs, N_j. This is only useful if more processors can be added, ideally having $N_p = N_j$; if the N_j is increased without increasing N_p, the added communication overhead will increase the elapsed time.

Factors in Choosing a Parallel Programming System

The authors considered three basic types of parallelization, which they refer to as vector computing, multi-processor computing, and distributed computing. As mentioned in the Introduction, vector computing is the parallel computation of groups of just a few arithmetic calculations. Although the programmer can assist in this by proper use of arrays, array indices, and `DO` loops, most work is done by the compiler. Since only a few small operations are done in parallel at a time, the problem must be very fine-grained, such as the computations used in solving systems of linear equations. The first author made a simple test of the hydrodynamics model on the San Diego Supercomputing Center's Cray Y-MP and was disappointed to find it vectorized very little, resulting in a speedup of only 4 to 5 times faster than a Sun 4/330. Given the hourly charges associated with using a supercomputer, it was decided not to pursue vectorization as a means of speeding the calibration.

Multiple-processor computing is that performed on computers with multiple processors, either hundreds or thousands of processors (massively parallel), or just a few processors. In either case special hardware and compilers are necessary, and for that reason was not considered, although the authors think this has considerable potential, since communications overhead can be kept low, yet large tasks can be parallelized, thus allowing a range of fine- to coarse-grained applications.

Since the Delta Modeling Section already had a network of single-processor Unix workstations, it was very desirable to use this existing computing power rather than buying an expensive, dedicated compute server. This led to considering only software systems which would run on the Section's existing machines in a distributed computing mode. This decision matched well with the nature of the problem, and its very high computational cost, which allowed the authors to disregard the high communication overhead associated with passing data over a network.

The following factors were considered in choosing a system to distribute the serial program:

1) Was the system only directly usable from the C language, or also from Fortran?

2) Would the system be easy to learn and use?

3) What was the price of the system?

4) How attentive would the maker be in answering questions and bug reports?

5) Was there some provision for fault tolerance in the system?

6) What was the communications overhead incurred by the system in passing data between jobs?

ISIS was chosen based on the following:

1) Usable directly from Fortran. Other possibilities, such as Remote Procedure Calls (RPC), were callable only from C, necessitating a number of C-to-Fortran and Fortran-to-C interface procedures.

2) Other systems such as parallelizing Fortran compilers would be easier to use, but were limited to a particular vendor's system, thus nullifying the benefit of using existing hardware.

3) Available at no cost through the Internet. This allowed it to be tested without risk to project funds, and avoided the lengthy purchasing bureaucracy.

4) The developers were readily available through electronic mail, and willing to respond quickly to bug reports. This was quite important when a couple of bugs were found which at first prevented the use of ISIS from Fortran.

5) Fault tolerance could be programmed. This has been taken advantage of to increase the reliability of the calibration, which method is described later.

6) ISIS offers a low-overhead method of communication called the bypass protocol, selected at compile time. A low communications overhead offers the possibility of medium- or fine-grain parallelization, especially on multi-processor computers.

What is the ISIS Toolkit?

The ISIS toolkit (or simply ISIS) was developed at Cornell University by Kenneth Birman and others (Birman, et al, 1990). It was designed to provide reliable (fault-tolerant) distributed systems along with a simplified programming paradigm, that of virtual synchrony (Birman and Cooper, 1990), and has been used in applications as varied as seismological monitoring systems, distributed file systems, etc. It is available via anonymous FTP from the Internet at ftp.cs.cornell.edu. The latest version is now sold commercially.

Steps Taken to Parallelize the Serial Program

First, a test system was created to experiment with ISIS. The test system has a dummy computational subroutine instead of the actual routine used to calculate the first derivatives, but otherwise is identical to the system used in practice. The test system consists of two separate programs known as the client and the server. The server program is run on each machine on the network, and consists of the computational routine and ancillary routines which create a server group and detect servers leaving and joining the group, notifying the client when they do so. The client controls the servers, sending them messages which controls which jobs they do, and collects the computed answers from them. It also keeps track of servers by receiving messages when servers are added or deleted from the server group.

Either the client or the servers may be started first. The first server to start creates a group for the servers, and subsequent servers simply join the existing group. When the client starts, it issues a call to the server group to see how many servers are initially available. Jobs (in our application, first derivative calculations) are then handed out to the servers and the client waits for all jobs to finish. If more jobs exist than available servers (the usual case), as a server finishes one job it is given another. If a server dies (probably because the machine it was running on crashed or was rebooted), the client is notified by another server as to which server died and which job was lost. The lost job is then restarted from the beginning on the next available server. If a new server is added (perhaps by a machine coming back on-line), the client is again notified and the new server given a job. So long as the client stays alive, and at least one server at any time, the run will continue.

The following changes were needed to the serial model before it could be run in parallel:

1) At the appropriate places the Fortran `DO` loops and calls to the model were replaced with ISIS calls to the job issue routine. Here, the ISIS call from the client to the server would normally wait until the server finished. However, since in the particular case of the calibration the server might take up to a few hours to return, the ISIS calls return immediately.

2) An ISIS wait call to wait for all jobs to finish was inserted where the end of the `DO` loop was. This was necessary because of the immediate return in step 1.

3) All output file names were replaced with names unique to each machine on the network. This is easily done in Unix by using environment variables for the filenames, passing the names in from the script file which starts each server on each machine.

All changes were done using the C pre-processor utility CPP, so that the code could be compiled in either a distributed or serial mode.

Appendix I contains sample code fragments before and after the parallelization.

Benefits of Distributed Computing

An obvious benefit, and the goal of the project, was to reduce the elapsed time needed to obtain a calibration of either the hydrodynamics or quality model. In the calibration, one base run of the model is performed with no parameters perturbed before the first derivative calculations can be done. Therefore, the serial time needed to complete a calibration is given by

$$E_{sc}=(L+1)E_iI \qquad \textbf{(13)}$$

where $\boldsymbol{L}$ is the number of parameter groups, $\boldsymbol{E_i}$ is the elapsed time for one iteration, and $\boldsymbol{I}$ the number of iterations.

In this application the communications overhead is low compared to the computational overhead, so that $p_c \approx 0$. Also, $N_j = L+1$ and $f = \frac{L}{L+1}$. Thus, the speedup of the calibration is given by

$$S_c = \frac{1}{1 + \frac{L}{L+1}(\frac{1}{N_p} - 1)} \quad \textbf{(14)}$$

Another benefit is the option of obtaining a better quality calibration. For instance, suppose that the calibration is done using 10 groups. Normally this would take 11 hours to do per iteration if each computation required 1 hour each. Using Eq. 14, and assuming 5 machines available of equal performance, a distributed calibration would then be done 3.67 times as fast as a serial calibration, finishing in 3 hours. However, it might be decided to increase the length of the calibration period 50% to obtain a greater range of measured values to calibrate against. Also, the number of groups might be increased, perhaps doubled, to have less channels per group, and thus increasing the fineness of the calibration. Then the parallel calibration time will increase by 2.5 times, but will still take less time (7.5 hours) than the original serial case. Obviously some trade-off is possible between speed increase and calibration quality increase, depending upon the user's preferences.

Finally, a more reliable calibration run is possible; that is, the probability of obtaining a valid run to completion is more likely. This can be seen by computing the likelihood that a particular run will succeed, that is, run to completion. Assume that machine failures are random events, with a mean time between failure T_f measured in days. Let the run time of the serial calibration E_{sc}, also be in days. The probability of a calibration run completing for the normal, non-distributed case is given by

$$P_s = (1 - \frac{1}{T_f})^{E_{sc}} \quad \textbf{(15)}$$

where P_s is the probability of success for the serial case. For the parallel case, the calibration run will continue if the machine the client is on does not fail and if at least one server machine is running at any time. Individual servers, and groups of servers, may be killed and the calibration will continue. A reasonable assumption is that if servers die they restart almost immediately, so that server downtime is not an issue. In addition, it is assumed that the probability that all servers die simultaneously and independently is nil; if all servers die, it would probably be due to some common event (e.g. building power outage) which would also affect the client, which probability of failure is already accounted for. In this case, the probability of success for the parallel case, P_p, is given by:

$$P_p = (1 - \frac{1}{T_f})^{E_{pc}} \quad \textbf{(16)}$$

with $E_{pc} = \frac{E_{sc}}{S_c}$. Figure 4 shows a comparison of success probabilities for both the serial and parallel cases for typical values of T_f, N_p, and L.

Since essentially only the failure of the client can cause a failure of the run, and since the client runs S_c times faster in the parallel mode, it is simply exposed to less chance of failure, and is thus more reliable. In fact, it is possible to make copies of the client program run on different machines, rendering the entire system almost 100 percent reliable. All clients would receive results from the servers performing the computations, but just one client at a time would be designated to reply to the servers. If that client died, another client would be selected automatically from the remaining clients and would carry on the task.

Future Work

Two tasks which could be done quickly are to implement the multiple client concept so as to improve reliability, and to add code so that the client has some notion of the speed of the servers so as to give jobs to the faster servers.

Another, more difficult task is to investigate ways of either reducing communication overhead or re-working the algorithms of the models so that more fine-grained tasks could be parallelized. For instance, it might be possible to parallelize the run of a model by itself, apart from the calibration procedure, to obtain faster production runs of the models, if each computational task took longer to do than the cost of communicating between servers. This would allow a group to better use existing computing resources, incrementally add to their computing pool, and increase reliability.

Summary

A Fortran application program which used to take many days to run on a single machine has been parallelized to run on a network of machines using the ISIS toolkit. Advantages, in addition to faster turn-around times, are a more accurate run and less chance of an incomplete run. The method is general, so that any application which meets the two requirements of, first, non-dependency between parts, and secondly, high computational overhead to communication overhead, could be parallelized in a similar manner. As more computers are added to the network the program run time is incrementally shorter.

Notes

Disclaimer: Mention of commercial products and companies does not imply endorsement by the State of California, or the authors or their employers past or present, and is done so for informational purposes only. Opinions expressed in this paper are those of the authors, not of their employers. ISIS is a trademark of Isis Distributed Systems, Inc.

The first author may also be contacted by email at rfinch@water.ca.gov; the second author at kao@capmkt.com.

Acknowledgements

The writers wish to thank George Barnes, Principal Engineer, Dept. of Water Resources, and Dr. Francis Chung, Supervising Engineer, Dept. of Water Resources, for supporting this project. ISIS was developed at Cornell University with financial support from the Defense Advanced Research Projects Agency.

Appendix I. Sample Fortran Code

Sample code fragment before parallelization:

```
 do ip=1,ipar
   dp=tst*param(ip)
   de=tst*eta(ip)
   prev_param=param(ip)
   param(ip)=param(ip)+dp
   call search(0)
   param(ip)=prev_param
   do iv=1,nv
     do im=1,imeas(iv)
       do it=1,nsamp(im,iv)
         gr(im,iv,p,it)=(vcalc(im,iv,it)-vp(im,iv,it))/de
       enddo
    enddo
  enddo
 enddo
```

The same task done in parallel. First the call in the client to the job issue routine in the client:

```
 num_jobs=ipar
 nreplies=bcast(g_addr_client, job_issue_entry, "0", 0)
 do while (.not. finished)
   call isis_sleep(15)
 enddo
```

The job issue routine issues calls to the servers as shown in this fragment:

```
 dp=tst*param(i)
 prev_param=param(i)
 param(i)=param(i)+dp
 nreplies=bcast(server_addr,server_execution_entry,
&     "%d%d%d%D%D%F", search_arg, i, ichn,
&     chan, nnc, group, nnc, param, nnp, 0)
 param(i)=prev_param
```

The server execution subroutine in the server program makes the actual call to the computational routine:

```
 call msg_get(msgp, "%d%d%d%D%D%F", search_arg, job_no,
&     ichn, chan, n_chngrp, group, n_chngrp, param, n_param)
 call search(0)

C Look for client and return the values
 gaddre = pg_lookup("client_pep")
 nreplies=bcast(gaddre, job_complete_entry, "%d%d%F",
&    job_no, my_uniq, vcalc, vcalc_size, 0)
```

Finally, the client program accepts the values returned by the server:

```
 call msg_get(msgp, "%d%d%F", job_no, server_uniq,
&     vcalc, vcalc_size)

 ip=job_no
 de=tst*eta(ip)
 do iv=1,nv
   do im=1,imeas(iv)
     do it=1,nsamp(im,iv)
       gr(im,iv,ip,it)=(vcalc(im,iv,it)-vp(im,iv,it))/de
     enddo
   enddo
 enddo

C Check if all jobs are done
 if (num_job_done .eq. num_jobs) then
    finished=.true.
 endif
```

Appendix II. References

Amdahl, G. (1967). "The validity of the single processor approach to achieving large-scale computing capabilities", *AFIPS Conference Proceedings* 30, 483-485.

Birman, K., and Cooper, R. (1990). "The ISIS Project: Real experience with a fault tolerant programming system", ISIS paper TR90-1138, Cornell University.

Birman, K., and Joseph T. (1987). "Exploiting Virtual Synchrony in Distributed Systems", *11th Association for Computing Machinery Symposium on Operating Systems Principles*, December 1987.

Birman, K., et al (1990). "The ISIS System Manual, Version 2.0", Cornell University.

Bulletin 160-87 (1987). "California Water: Looking to the Future", California Department of Water Resources, November 1987.

Buzbee, B., (1986). "A strategy for vectorization", *Parallel Computing* (3), 187-192.

Evenson, D., and Johnson, A. (1975). "Parameter Estimation Program. Program Documentation and User's Manual", Water Resources Engineers, Inc.

Fischer, H.B. (1984). "Fischer Delta Model, Volume 2: Reference Manual", Report HBF-84/01, Hugo B. Fischer, Inc.

Hack, J. (1989). "On the promise of general-purpose parallel computing", *Parallel Computing* (10), 261-275.

Karplus, W.J. (1983). "The Spectrum of Mathematical Models", *Persepectives in Computing*, May 1983, 4-13.

WEF (1989). "Layperson's Guide to the Delta", Water Education Foundation, 1989.

Appendix III. Notation

The following symbols are used in this paper:

E_p = elapsed time of parallelized program;

E_p' = elapsed time of a parallel task;

E_{pc} = elapsed time of a parallel calibration;

E_{sc} = elapsed time of a serial calibration;

E_s = elapsed time of serial program;

f = fraction of program that can be parallelized;

$\bar{h}$ = computed variables;

$\hat{h}$ = measured variables;

I = number of iterations;

J = Jacobian matrix;

K = parameters to be calibrated;

L = number of parameter groups;

N = number of computed variables;

N_p = number of processors;

N_j = number of jobs;

P = performance function;

P_p = probability of completion for parallel program;

P_s = probability of completion for serial program;

p_c = processor time for communication;

p_j = processor time of a parallel job;

S = speedup;

T_f = mean time between failure;

ϵ = calibration errors at each time and location;

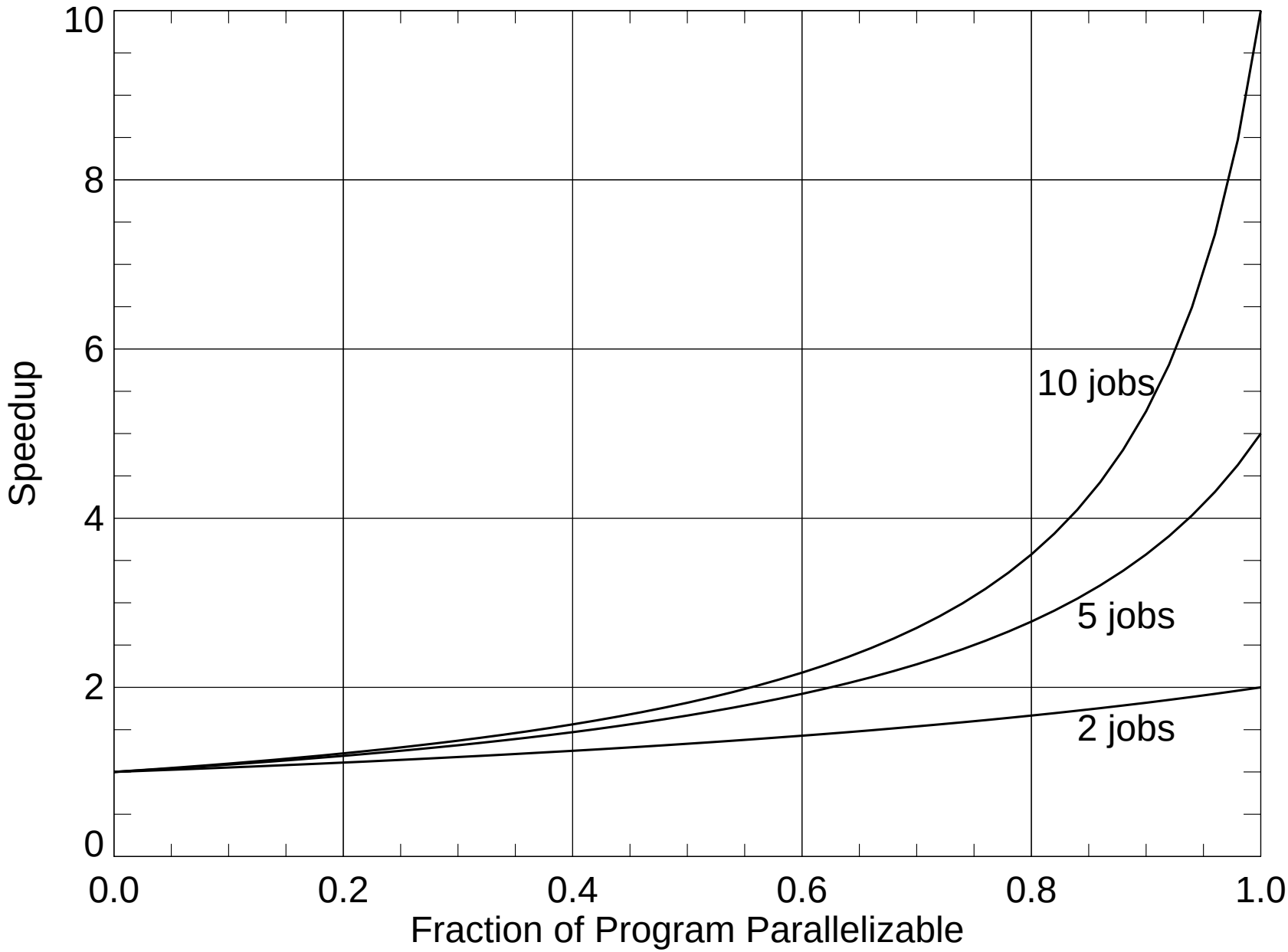
τ = number of measurements in time;

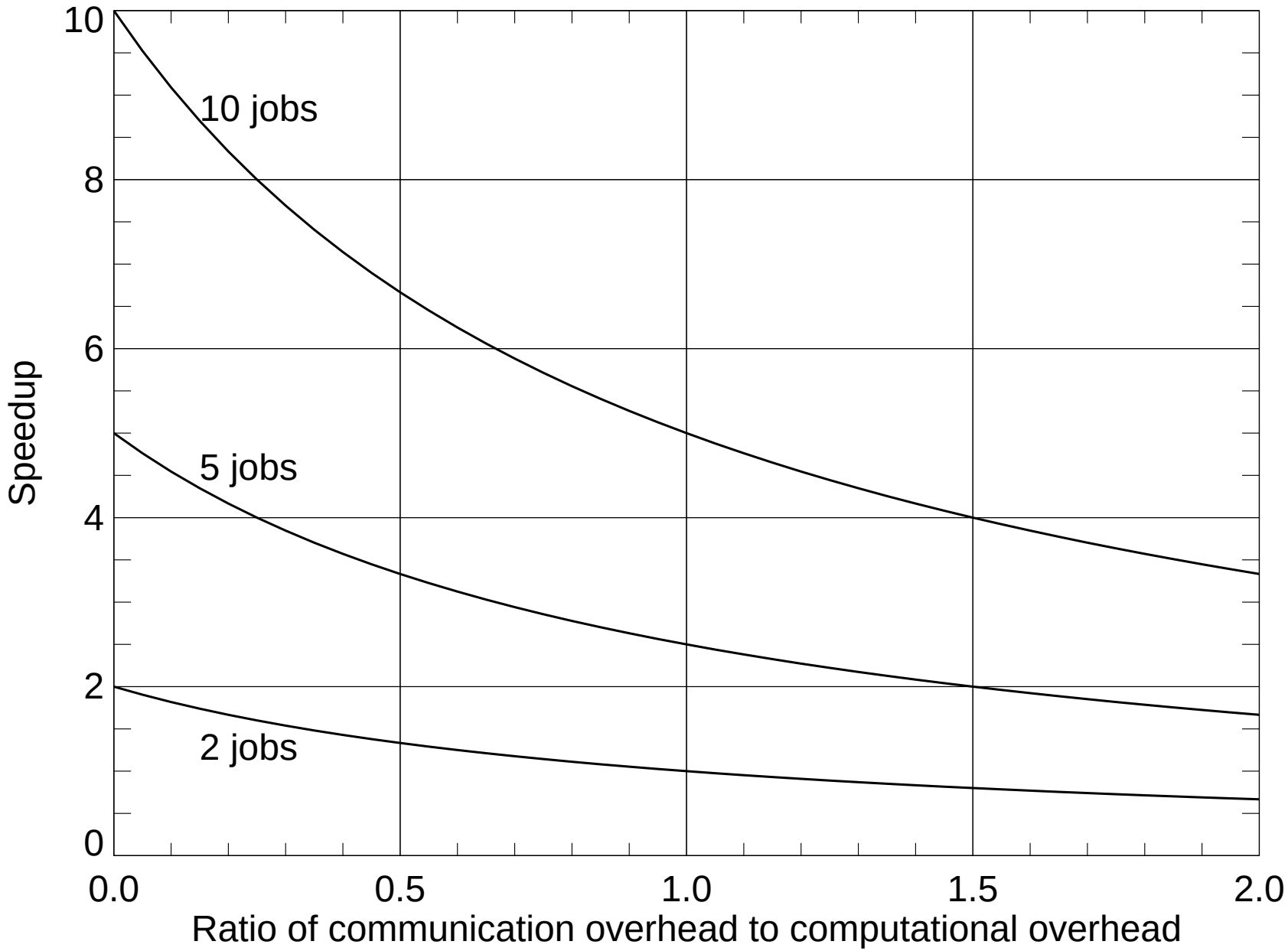
Reprint sales summary.

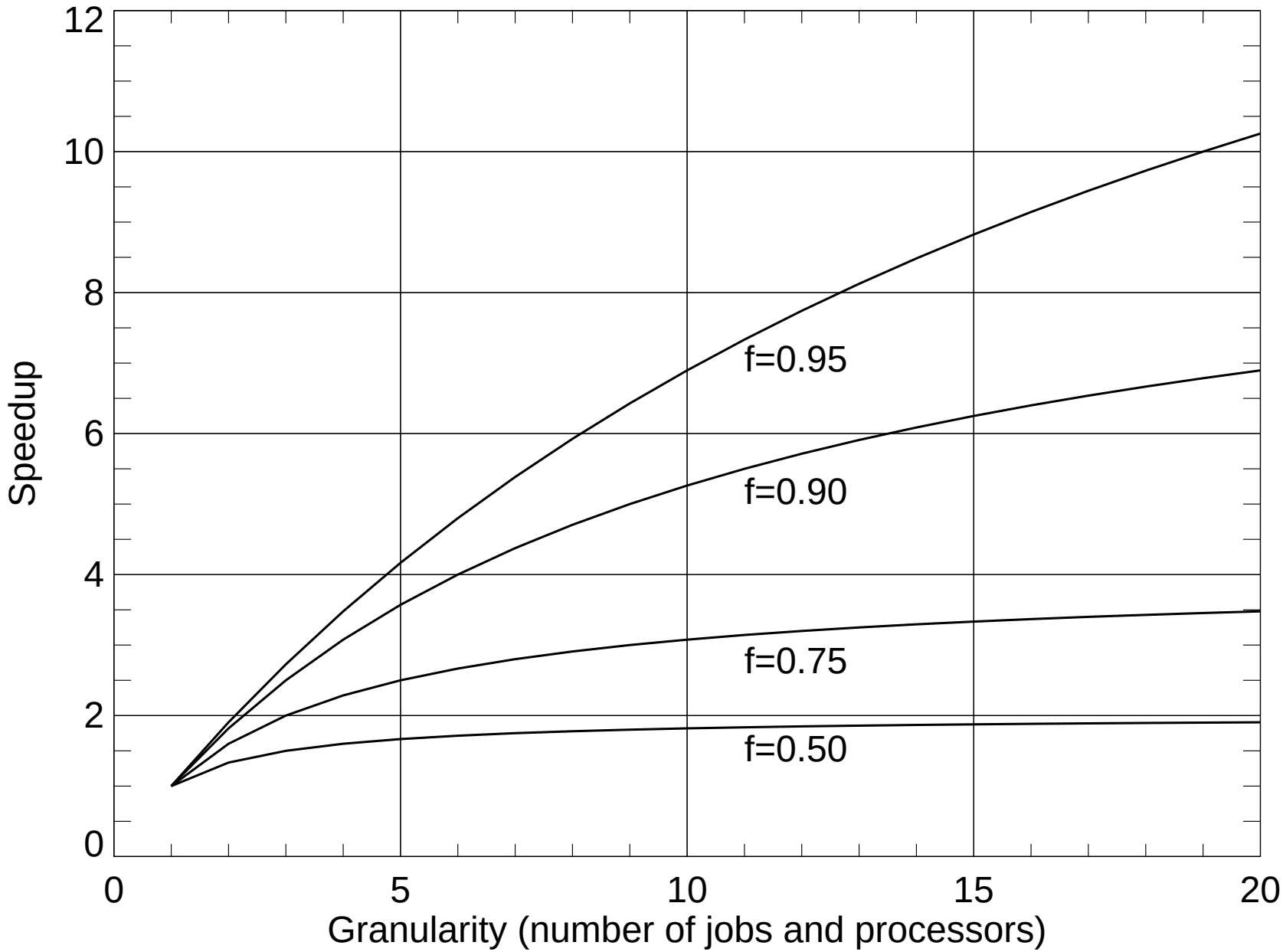
The paper describes a method of parallelizing a serial programs written in Fortran, using the ISIS toolkit. Considerations for parallelizing existing programs are given.

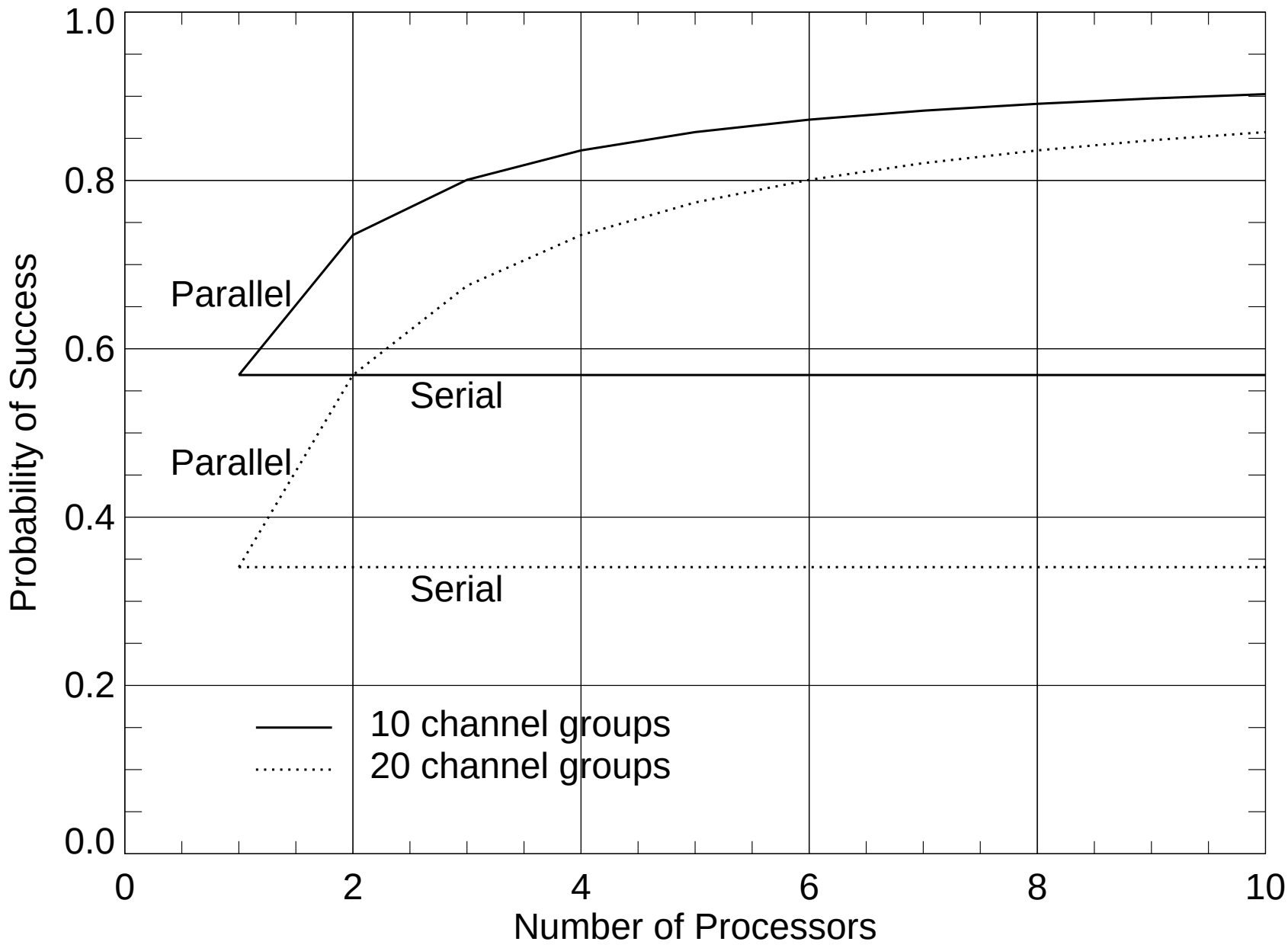
Keywords

Computers, distributed computing, parallel computing, calibration, mathematical modeling, Fortran.


10
8
6
4
2
0
Speedup
10 jobs
5 jobs
2 jobs
0.0
0.2
0.4
0.6
0.8
1.0
Fraction of Program Parallelizable


10
8
6
4
2
0
Speedup
10 jobs
5 jobs
2 jobs
0.0
0.5
1.0
1.5
2.0
Ratio of communication overhead to computational overhead


12
10
8
6
4
2
0
Speedup
f=0.95
f=0.90
f=0.75
f=0.50
0
5
10
15
20
Granularity (number of jobs and processors)


1.0
0.8
0.6
0.4
0.2
0.0
Probability of Success
Parallel
Serial
Parallel
Serial
10 channel groups
20 channel groups
0
2
4
6
8
10
Number of Processors